This paper appeared in: ACM SIGPLAN Notices, 28(8):99–108, August 1993

Compiling Machine-Independent Parallel Programs

Michael Philippsen, Ernst A. Heinz, Paul Lukowicz
Universität Karlsruhe
Fakultät für Informatik
D-7500 Karlsruhe, F.R.Germany
email: phlipp@ira.uka.de

Abstract

Initial evidence is presented that explicitly parallel, machine-independent programs can automatically be translated into parallel machine code that is competitive in performance with hand-written code.

The programming language used is Modula-2*, an extension of Modula-2, which incorporates both data and control parallelism in a portable fashion. An optimizing compiler targeting MIMD, SIMD, and SISD machines translates Modula-2* into machine-dependent C code.

The performance of the resulting code is compared to that of equivalent, carefully hand-coded and tuned programs. On a MasPar MP-1 (SIMD machine with up to 16k processors) the Modula-2* programs typically achieve 80% of the performance of the hand-coded parallel versions. When targeting sequential processors, the Modula-2* programs reach 90% of the performance of hand-coded sequential C. (There are no MIMD results yet.)

The effects of two major optimization techniques, synchronization point elimination and data/process alignment are also quantified.

1 Introduction

Effective programmability of parallel machines is one of the most pressing problems in parallel computing. The key aspect of this problem is efficiency-preserving portability.

The first major concern, *portability*, is as essential for parallel computing as it is for sequential computing: one simply cannot afford to rewrite parallel programs for each machine. Portability can be achieved with machine-independent programming languages that allow clear expression of parallel algorithms and are free of hardware quirks that may differ from one computer to the next.

The second major concern is *efficiency*. Programs expressed in a high-level, portable language must be compilable into parallel machine code of satisfactory efficiency on a wide range of architectures. Efficiency is satisfactory if the compiled code approaches the performance of hand-tuned machine-dependent code.

This paper is primarily concerned with efficiency. It provides a quantitative evaluation of the code produced by a compiler for a high-level, portable programming language with explicit parallelism. The language is Modula-2*, an extension of Modula-2. The extensions are small and could be incorporated into other imperative languages, including Fortran. At present, the compiler targets the MasPar MP-1 series (large scale SIMD systems), LANs (medium scale MIMD systems), and sequential workstations (SISD systems). Measurements of a set of benchmarks support the

Hypothesis: Explicitly parallel and machine-independent programs can automatically be translated into machine-dependent parallel code that is competitive in performance with optimized hand-written code.

This result is important for writing explicitly parallel programs and for converting existing sequential programs to parallel ones. With good compilers, the manual conversion of a sequential program can concentrate on finding parallel algorithms, while ignoring machine-dependent details. The necessary mapping to a given machine architecture is performed completely automatically. The advantage of this separation of concerns is not only that it simplifies the conversion process, but it also assures that the result of the conversion is a machine-independent program that can be run on different machines after recompilation.

Furthermore, we present evidence that a compiler can also produce highly efficient *sequential* code from parallel programs. Sequential efficiency is important for several reasons. First, it allows programmers to use parallel language constructs even when targeting sequential machines. Parallel constructs free programmers from the task of manually sequentializing an algorithm where parallel expression is more natural. Second, parallel programs can be developed and tested on sequential machines without incurring unjustifiable overhead. Finally, the fact that a compiler for parallel machines produces efficient sequential code when setting the number of processors to unity provides a good indication about the generality and scalability of the code generation techniques employed.

In section 2, we briefly introduce Modula-2* while the main features of our Modula-2* System (compiler, debugger, libraries, runtime system) are described in section 3. We present the benchmarks, experiments, and their results in section 4 and conclude with a discussion of the quantitative effects of two major optimization techniques.

2 Modula-2*

The programming language Modula-2* was developed to allow for high-level, problem-oriented and machine-independent parallel programming. As described in [19], it provides the following features:

- An arbitrary number of processes operate on data in the same single address space. Note that shared memory is not required; a single address space merely permits all memory to be addressed uniformly, but not necessarily at uniform speed.
- Synchronous and asynchronous parallel computations as well as arbitrary nestings thereof can be formulated in a totally machine-independent way.
- Procedures may be called in any context (sequential, synchronous, or asynchronous) and at any nesting depth. Furthermore, additional parallel processes can be created inside procedures (recursive parallelism).
- All the abstraction mechanisms of Modula-2 are available for parallel programming.

Modula-2* extends Modula-2 with the following two language constructs.

1. The `FORALL` statement, which has a synchronous and an asynchronous version, is the only way to introduce parallelism into a Modula-2* program.
2. The distribution of array data is optionally specified by *allocators*, e.g. `SPREAD`, `CYCLE`. They do not have any semantic meaning and are just layout hints for the compiler.

Because of the compactness and simplicity of the extensions, they could easily be incorporated into other imperative programming languages, such as Fortran, C, or Ada. In Modula-2* the syntax of the `FORALL` statement is:

```
ForallStatement =
FORALL ident ":" SimpleType IN (PARALLEL | SYNC)
[VarDecl+ BEGIN]
  StatementSequence
END.
```

SimpleType is an enumeration or a possibly non-static subrange, i.e. the boundary expressions may contain variables. The `FORALL` creates as many (conceptual) processes as there are elements in *SimpleType*. The identifier introduced by the `FORALL` statement is local to it and serves as a runtime constant for every process created by the `FORALL`. The runtime constant of each process is initialized to a unique value of *SimpleType*. The `FORALL` statement provides an optional section for the declaration of variables local to each process. These local variables lead to better source code structuring, thus greatly increasing the readability and efficiency of parallel code.

Each process created by a `FORALL` executes the statements in *StatementSequence*. The `END` of a `FORALL` statement imposes a *synchronization barrier* on the participating processes: termination of the whole `FORALL` statement is delayed until *all* created processes have finished their execution of *Statement-Sequence*.

The version of the `FORALL` statement (synchronous or asynchronous) determines whether the created processes execute *StatementSequence* in lock-step or concurrently.

Hence, for non-overlapping vectors `X`, `Y`, and `Z` the following asynchronous `FORALL` statement suffices to implement the vector addition `X := Y + Z`.

```
FORALL i : [1..N] IN PARALLEL
  X[i] := Y[i] + Z[i]
END
```

In contrast to the above, parallel modifications of overlapping data structures may require synchronization. Thus, irregular data permutations can be implemented as follows:

```
FORALL i : [1..N] IN SYNC
  X[i] := X[p(i)]
END
```

This program permutes the vector `X` according to the permutation function `p`. The semantics of the synchronous `FORALL` ensure that all rhs elements `X[p(i)]` are read and temporarily stored before any lhs variable `X[i]` is written.

The behavior of branches and loops inside synchronous `FORALL`s is defined with an MSIMD (multiple SIMD) machine in mind. This means that Modula-2* does require any synchronization between different branches of synchronous `CASE` or `IF` statements. The exact synchronous semantics of all Modula-2* statements are defined in [19].

The synchronous version of this `FORALL` operates much like the HPF `FORALL`, except that it is fully orthogonal to the rest of the language: Any statement, including conditionals, loops, other `FORALL`s, and subroutine calls may be placed in its body. Thus, the language explicitly supports nested and recursive parallelism. There is no concept of asynchronous parallelism in HPF.

3 The Modula-2* System

The Modula-2* System currently targets the MasPar MP-1 series of massively parallel processors (SIMD), heterogenous LANs of Unix workstations (MIMD), and single standard Unix workstations (SISD). The Modula-2* System consists of

1. an optimizing and restructuring compiler,
2. a machine-dependent runtime system,
3. libraries of scalable parallel operations (enumeration, reduction, scan, etc.),
4. a parallel debugger.

Below, we describe each part of the Modula-2* System in some detail.

3.1 Compiler

General Architecture. To keep major parts of the compiler machine-independent, Modula-2* programs are translated to a general intermediate representation. Based on a study of different parallel machines, we decided to use C augmented with a set of macros as an intermediate language [13]. Macros are expanded using target-specific include files yielding the appropriate parallel C derivate. Thus, retargeting the compiler only requires the exchange of the macro package and some libraries.

Optimizations. On parallel machines, optimizations tend to improve program runtime dramatically. Therefore, the Modula-2* compiler performs various optimizations and code restructurings summarized below (for more details see [17]). In the following subsections, we briefly sketch the main optimizations that are implemented in our Modula-2* compilers. In section 4.4 we show the quantitative effects of these techniques.

Automatic Data and Process Distribution

On distributed memory machines, the distribution, i.e. alignment and layout of data and processes over the available processors is a central problem.

Alignment is the task of finding an appropriate tradeoff between the two conflicting goals of (1) data locality and (2) maximum degree of parallelism. Our automatic alignment algorithm is descibed in [16] and briefly sketched below by means of an example. *Layout* is the assignment of aligned data structures and processes to the available processors. Desirable goals are (3) the exploitation of special hardware supported communication patterns and (4) simple address calculations. We use an automatic mapping [15] of arbitrary multidimensional arrays to processors and thus exploit grid communication if available and achieve efficient address calculations.

To align arrays `A` and `B` of the following example, array `A` is enlarged and shifted to the left. All index expressions involved are transformed accordingly.

```
VAR A: ARRAY [1..90]  SPREAD OF INTEGER;
    B: ARRAY [0..100] SPREAD OF INTEGER;

FORALL i:[1..90] IN SYNC
  A[i] := B[i-1];
  B[i] := 0
END
                        ↓

VAR A,B : ARRAY [0..100] SPREAD OF INTEGER;

FORALL i:[1..90] IN SYNC
  A[i-1] := B[i-1];
  B[i]   := 0
END
```

The shift leads to the same index expressions on a per statement basis. The enlargement decouples alignment and layout. Since the resulting arrays have the same size, the layout algorithm maps corresponding elements of the array to the same processor. We allow for moderate storage waste because the primary goal is execution speed.

Up to now we have only dealt with the data alignment. Process alignment is also achieved by means of a source-to-source transformation. During this transformation, the `FORALL`s are attributed with an `ALIGNED WITH` clause that directs the code generator to allocate each process where the corresponding data element resides:

```
VAR A,B : ARRAY [0..100] SPREAD OF INTEGER;

FORALL i:[0..89] IN SYNC ALIGNED WITH A[i]
  A[i] := B[i]
END;
FORALL i:[1..90] IN SYNC ALIGNED WITH B[i]
  B[i] := 0
END
```

The original `FORALL` has been split into two parts. In both `FORALL`s the process with index `i` will be executed where data element `B[i]` resides, resulting in local accesses. Local accesses could not be achieved with a single `FORALL`.

Elimination of Synchronization Barriers

The semantics of synchronous `FORALL`s in [19] require a vast number of synchronization barriers. Most real synchronous `FORALL`s, however, only need a fraction thereof to ensure correctness [9]. Redundant synchronization barriers can be detected with data dependence analysis [22, 3].

To understand the techniques of automatic synchronization barrier elimination consider the synchronous `FORALL` statement below, followed by two possible translations.

```
FORALL i: [1..N] IN SYNC
  Z[i] := Z[i+1];
  X[i] := X[2*i];
  Y[i] := Y[p(i)];
END
```

```
FORALL i:[1..N] IN PARALLEL
  H1[i] := Z[i+1]
END;
FORALL i:[1..N] IN PARALLEL
  Z[i] := H1[i]
END;     FORALL i:[1..N] IN
FORALL i:[1..N] IN PARALLEL
  H2[i] := X[2*i]
END;
FORALL i:[1..N] IN PARALLEL
  X[i] := H2[i]
END;
FORALL i:[1..N] IN PARALLEL
  H3[i] := Y[p(i)]
END;
FORALL i:[1..N] IN PARALLEL
  Y[i] := H3[i]
END
```

```
FORALL i:[1..N] IN PARALLEL
  H1[i] := Z[i+1];
  H2[i] := X[2*i];
  H3[i] := Y[p(i)]
END;
PARALLEL
  Z[i] := H1[i];
  X[i] := H2[i]
  Y[i] := H3[i]
END;
```

The translation on the left shows an equivalent program, in which all synchronization points appear at the end of asynchronous `FORALL`s. The compiler detects that four of the six synchronizations are redundant and restructures the code accordingly. The optimized result is shown on the right.

Known Weaknesses

Currently, the compiler does not exploit the possibility of grid communication on the MasPar. Non-local data is accessed with general communication. Although the necessary information is present in the compilation process this is not yet implemented.

On MIMD machines with high latency networks the following optimizations, which are not implemented yet, will improve performance: The combination of messages that have the same source or destination will lead to larger packets and less total latency. With pre-fetch or post-store analysis Computation and communication can be overlapped to hide remaining latency.

Furthermore, there are some performance problems when translating nested parallelism. Work on better scheduling strategies is in progress.

3.2 Runtime System

The Modula-2* runtime system performs the initialization, maintenance, and cleanup of code sections executed in parallel. Runtime system functions are provided by efficiently implementable, machine-independent macro interfaces.

The MasPar MP-1 series runtime system makes use of the MasPar system library. The LAN runtime system is built on top of p4 [4], a message passing parallel programming system available for a variety of machines. Therefore, we are able to target heterogenous LANs. The use of p4 should also make our LAN compiler a sound basis for a future MIMD Modula-2* compiler.

3.3 Parallel Libraries

The Modula-2* parallel libraries comprise reductions, scans and enumerations. They aim at scalability, portability, and efficiency of frequently used parallel operations. Scalability means that the library routines operate on open array parameters of arbitrary size. We ensure portability by providing the same interfaces on all target machines. To achieve efficiency, we exploit low-level features of each target machine in the different library implementations.

Another interesting feature of these libraries is their functional diversity. Wherever possible, normal, masked, segmented, and universal (masked plus segmented) versions of the parallel operations are provided.

3.4 Parallel Debugger

The Modula-2* source-level debugger [7] allows for visual interactive debugging under X-Windows. The central concepts of debugging parallel Modula-2* programs are process and data visualization. The debugger enables users to trace activities executed in parallel by providing abstraction mechanisms like grouping, parallel call trees, and simultaneous source code views in different windows. For data visualization, 2D-slices of multidimensional distributed arrays can be displayed graphically in so-called "visualizer windows". Furthermore, the debugger is able to collect rudimentary profiling data by counting statement or subroutine invocations.

4 Benchmarks and Results

At the moment, our benchmark suite consists of thirteen problems collected from the literature [1, 6, 11, 8, 5]. For each problem, we implemented the same algorithms in Modula-2*, in sequential C, and in MPL[1]. Then we measured the runtimes of our implementations on a 16K MasPar MP-1 (SIMD) and a SparcStation-1 (SISD) for widely ranging problem sizes. Measurements for LANs are not yet available because the tedious and error-prone task of implementing hand-coded versions is still in progress.

Modula-2* Programs. In Modula-2* we employ our libraries wherever possible. A technical deficiency in our current Modula-2* compiler forced us to manually "unroll" two-dimensional arrays into one-dimensional equivalents. This will no longer be necessary in the near future.

MPL Programs. In MPL we implemented the same algorithms as in Modula-2* and carefully hand-tuned them for the MasPar MP-1 architecture. The MPL programs make extensive use of local access, neighborhood communication, standard library routines, and other documented programming tricks. To ensure the fairness of the comparison, the resulting MPL programs are as generally scalable as their Modula-2* counterparts. Since scalability is not restricted to multiples of the number of processors, boundary checks are required in every virtualization loop.

Sequential C Programs. The sequential C programs implement the parallel algorithms on a single processor. We use optimized sequential libraries wherever possible.

In the following, we first compare the resource consumption of these three program classes. Then we discuss their overall performance and present each problem together with its specific performance results in some detail. In section 4.4 we show the quantitative effects of the optimization techniques.

4.1 Resource Consumption

The comparison is based on the criteria program space, data space, development time, and runtime performance.

Program Space. Our compiler translates Modula-2* programs via C plus macros to MPL or C. The result-

[1] MPL [14] is a data-parallel extension of C designed for the MasPar MP-1 series. In MPL, the number of available processors, the SIMD architecture of the machine, its 2D mesh-connected processor network, and the distributed memory are visible. The programmer writes a SIMD program and a sequential frontend program with explicit interactions between the two. MPL provides special commands for neighborhood and general communication. Virtualization loops and distributed address computations must be implemented by hand.

ing programs consume slightly more space than the hand-coded MPL or C programs. Regarding source code length, Modula-2* programs are typically half the size of their corresponding MPL or C programs.

Data Space. The memory requirements of the Modula-2* programs are typically similar to those of the MPL and C programs. Memory overhead, i.e. variable replication into temporaries, occurs during synchronous assignments. This replication, however, most often is also required in hand-coded MPL. Furthermore, there is some additional overhead involved in controlling synchronous, nested, and recursive parallelism (16 bytes per `FORALL`).

Development time. Due to compiler errors detected while implementing the benchmarks, we cannot give exact quantitative figures on implementation and debugging time. However, we estimate that the implementation effort in Modula-2* is a fifth of the MPL effort.

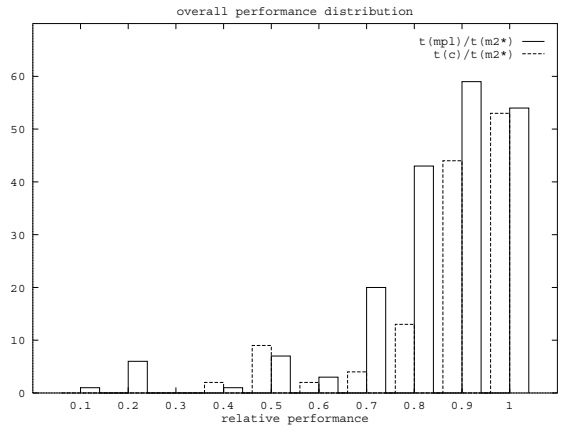
4.2 Runtime Performance

MPL versus Modula-2*. The general relative performance of Modula-2* is quite stable over all problem sizes and averages to 80%. Modula-2* typically achieves 70%–90%, with peaks at 100% of the MPL performance.

Sequential C versus Modula-2*. The general relative performance of Modula-2* is again quite stable over all problem sizes and averages to 90%. Modula-2* typically achieves 70%–90% of the sequential C performance, with peaks at 100%.

For widely varying problem sizes we measured the runtime of each test program on a 16K MasPar MP-1 and a SparcStation-1. We used the high-resolution DPU timer on the MasPar and the UNIX `clock` function on the SparcStation (sum of user and system time). Below, t_{m2*} represents the Modula-2* runtime on either a 16K MasPar MP-1 or a SparcStation-1 (as appropriate); t_{mpl} gives the MPL runtime on a 16K MasPar MP-1; t_c stands for the sequential C runtime on a SparcStation-1.

We define performance as problem size per time unit and focus on performances $\frac{size}{t_{m2*}} / \frac{size}{t_{mpl}} = t_{mpl}/t_{m2*}$ and $\frac{size}{t_{m2*}} / \frac{size}{t_c} = t_c/t_{m2*}$.



The overall distribution of relative performances proves to be encouraging. The above histogram provides the number of relative performance values falling into one of the classes [0%–5%), [5%–15%), ..., [95%–100%]. The numbers are the accumulated sums over all problems and problem sizes (all data points).

4.3 Benchmarks

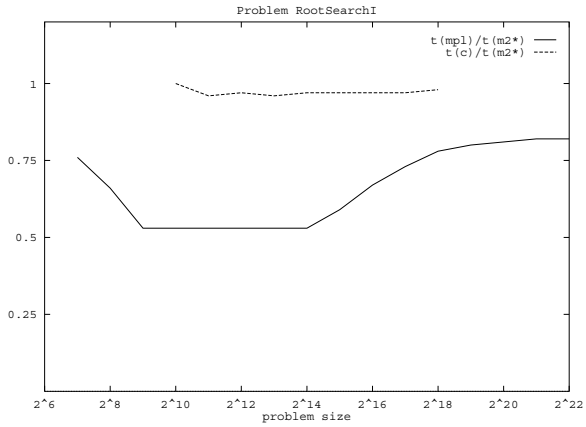
The benchmark suite consists of thirteen problems collected from the literature [1, 6, 11, 8]. The problems 4.3.11, 4.3.2 and 4.3.8 have been defined in [5] to test the expressive power of parallel programming languages. Some problems (4.3.1, 4.3.5, 4.3.12) are chosen from text books on parallel programming [1, 6, 11]. The problem of finding the longest common subsequence (4.3.3) is well known in text processesing and computational biology [18]. The remaining problems have been introduced by other authors and compiler groups [12, 8, 10]. The benchmark suite does not contain standard numeric operations since we are convinced that these routine will require low level library implementation which is unlikely to be done by an end user in Modula-2*.

In the problem descriptions below n is used as problem size that occurs in the graphs.

4.3.1 Root Search

Problem: Determine the value of $x \in [a, b]$ such that $f(x) = 0$, given that f is monotone and continuously differentiable.

Approach I: The problem is solved with multisection. The interval $[a, b]$ is equally divided over n processes. If f has a root in $[a, b]$ then there is exactly one process p with $f(x_{p-1}) \cdot f(x_p) < 0$. Update the interval $[a', b'] := [x_{p-1}, x_p]$. Iterate until the error $b' - a' < \epsilon$.

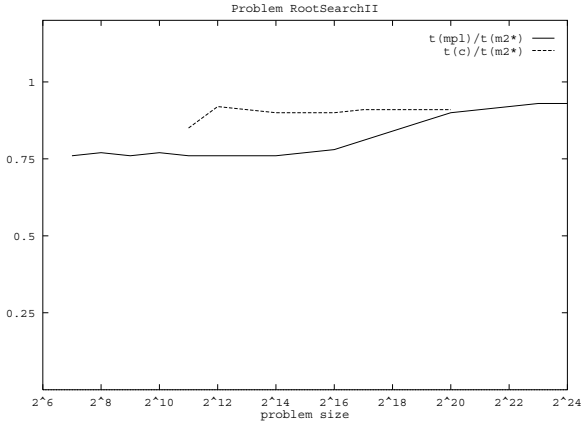


The main reason for the better runtimes of the hand-coded MPL program is the way neighboring data elements are accessed. The MPL program exploits the hardware supported XNET communication, whereas the Modula-2* compiler currently uses the much slower general communication. Global communication becomes slower with an increasing number of data packets in the network, whereas XNET performance is independent of the

load. Thus, the performance ratio drops initially, until general communication is saturated. With growing virtualization ratio ($> 2^{14}$), an increasing number of accesses to neighboring data elements is local in both the MPL implementation and the Modula-2* translation. Since the fraction of the overall runtime spent in communication shrinks, the performance of the Modula-2* program improves to 80%.

Approach II: Again, the interval $[a, b]$ is divided evenly over all processes. Then each process performs Newton's iteration. The algorithm terminates when a process finds the root.

Note: This problem occurs frequently in science and engineering applications [1].



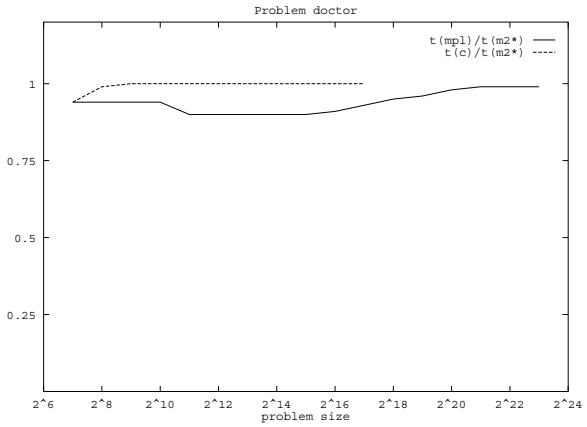
Since the implementations have total locality the performance is better than that of approach I. The Modula-2* compiler uses a general translation scheme for the `FORALL` statement that allows for nested parallelism. This generality, however, is more costly than the straightforward implementation of virtualization loops in the MPL program. For problem sizes $< 2^{14}$ the loops are iterated only once. For growing virtualization ratios, the loop overhead becomes smaller compared to the work done in all iterations, leading to growing performance of Modula-2*.

4.3.2 Doctor's Office

Problem: A set of n patients, a set of doctors, and a receptionist are given. The task is to model the following interactions: Initially, all patients are well and all doctors are in a FIFO queue awaiting sick patients. Then patients become sick at random and enter a FIFO queue for treatment by one of the doctors. The receptionist handles the two queues, assigning patients to doctors in FIFO manner. As soon as a doctor and patient are paired, the doctor diagnoses the illness and treats the patient in a random amount of time. After finishing with a patient, the doctor rejoins the doctor's queue to await another patient. The output of the problem is intentionally unspecified (from [5]).

Approach: The random amounts of time that patients are well and that doctors need to treat illnesses are counted down in parallel. The FIFO assignments of doctors to patients is done in parallel, too. The output is a list of timestamps, indicating when patients became ill, and list of pairings (doctor, patient, treatment time).

The curve of the MasPar performance is shaped similar to that of problem 4.3.1 (approach I). However, the amount of computation dominates the effect of communication operations.

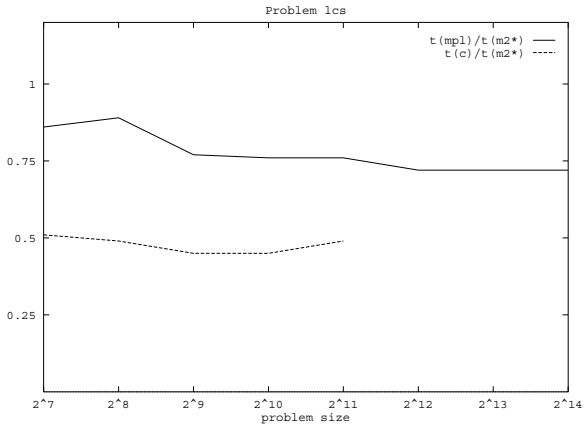


4.3.3 Longest Common Subsequence

Problem: Two strings $A = a_1 a_2 \cdots a_l$ and $B = b_1 b_2 \cdots b_m$ are given. Find a string $C = c_1 c_2 \cdots c_p$ such that C is a longest common subsequence of A and B. (C is a subsequence of A if it can be constructed by removing elements from A without changing their order. A common subsequence must be constructible from both A and B.)

Approach: The solution uses a wave-front implementation of dynamic programming. It causes intensive access to neighboring data elements. The problem size is $n = \max(l, m)$.

Note: The problem is presented in detail in [18]. The parallel solution is based on [2].



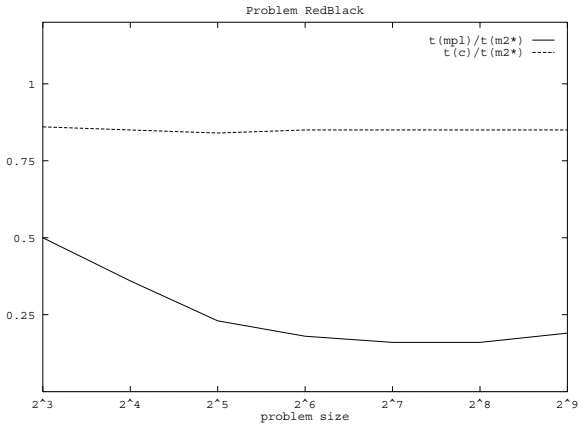
The curve of the MasPar performance is shaped similar to that of problem 4.3.1 (approach I). The effect of global versus XNET communication is smaller when few packets are sent (problem size $< 2^9$). Due to limited memory, only problem sizes smaller than 16k are considered. Thus, the expected performance growth for bigger problem sizes is not visible.

4.3.4 Red/Black Iteration

Problem: Implement a red/black iteration, i.e., the kernel of a solver for partial differential equations.

Approach: The implementation is straightforward. See

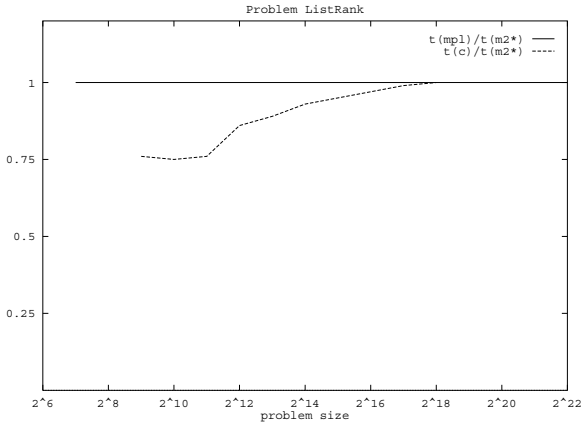
for example [1]. It almost exclusively references neighboring data elements ina $n \cdot n$-matrix.
Note: This problem often serves as a case study for implementors of automatically parallelizing compilers, e.g. [10].



See the explanation for problem 4.3.1 (approach I). The Red/Black problem is quadratic. Problem size 2^7 requires 2^{14} matrix elements and therefore corresponds to the machine size of the MasPar (16k).

4.3.5 List Rank

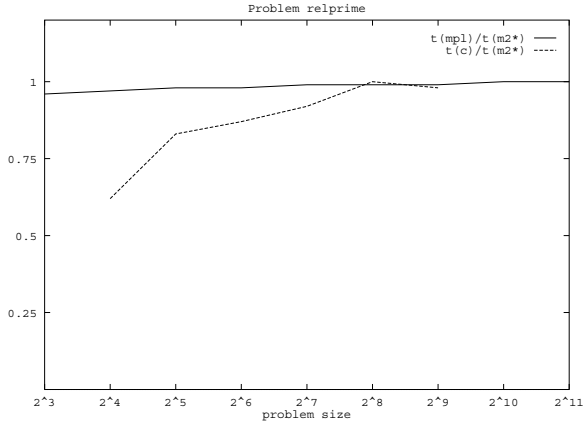
Problem: A linked list of n elements is given. All elements are stored in an array $A[1..n]$. Compute for each element its rank in the list.
Approach: This problem is solved by pointer jumping.
Note: Ranking the elements of a list is one of the elementary list processing tasks [11].



The good result on the MasPar is caused by the fact that both MPL and Modula-2* must use general communication.

4.3.6 Pairs of Relative Primes

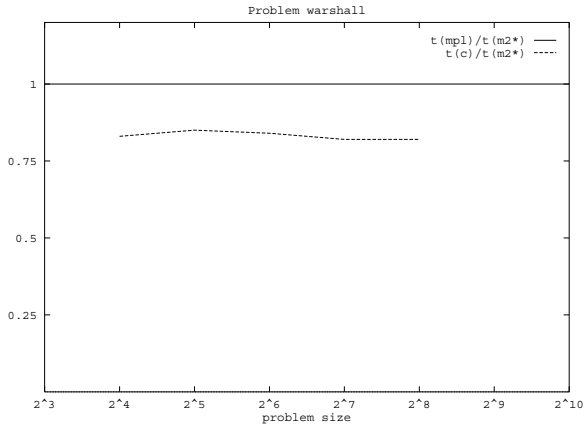
Problem: Count the number of pairs (i, j) with $2 \leq i < j \leq n$ that are relatively prime, i.e. the greatest common divisor of i and j is 1.
Approach: The solution is based on a data-parallel implementation of the GCD algorithm followed by an add-scan.
Note: The problem was suggested by Hatcher [8].



The parallel invocation of a GCD procedure with its parallel `while` construct is the dominant cost producer in this example. Since this is implemented almost identical in MPL and the Modula-2* version on the MasPar, the same runtimes can be measured.

4.3.7 Transitive Closure

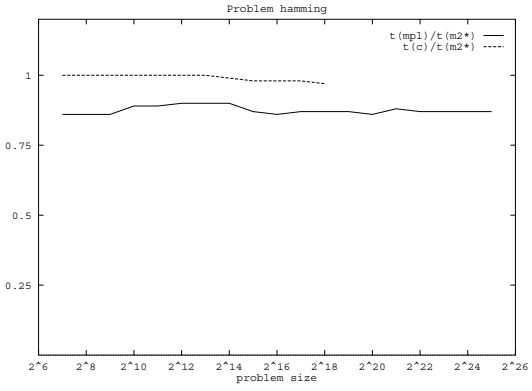
Problem: The adjacency matrix of a directed graph with n nodes is given. Find its transitive closure.
Approach: Process the adjacency matrix according to the property that if nodes x and m as well as nodes m and y are (transitively) adjacent, then x and y are (transitively) adjacent. The algorithm is due to Warshall [21].
Note: The problem was suggested by Hatcher [8].



The good result on the MasPar is caused by the fact that both MPL and Modula-2* must use general communication.

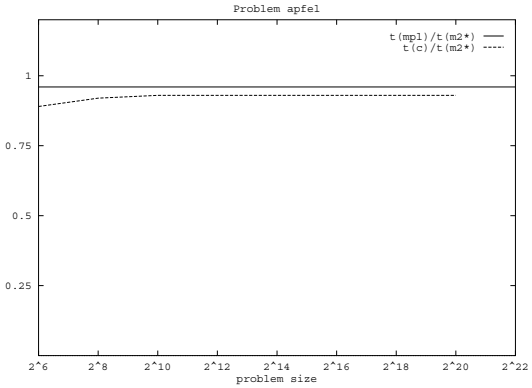
4.3.8 Hamming's Problem

Problem: A set of primes $\{a, b, c, \ldots\}$ of arbitrary size and an integer n are given. Find all integers of the form $a^i \cdot b^j \cdot c^k \cdot \ldots \leq n$ in increasing order and without duplicates.
Approach: For each given prime p compute the power set $\{p^i | p^i \leq n\}$. Combine any two power sets to a new one, while enforcing that the products remain $\leq n$. Repeat the combination for all power sets.
Note: The problem has been suggested in [5].



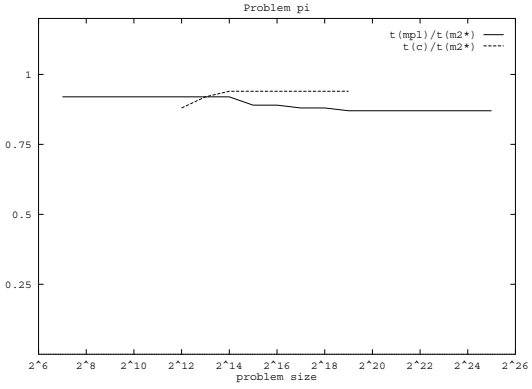
4.3.9 Mandelbrot Set

Problem: Compute the well-known Mandelbrot set.
Approach: Perform all iterations in parallel.
Note: Performance is excellent due to the absence of communication.



The good result on the MasPar is caused by the fact that both MPL and Modula-2* rely on total locality.

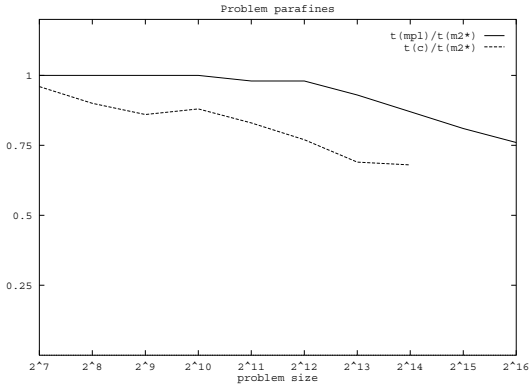
4.3.10 Estimation of Pi



Problem: Compute π using the equation $\pi = \int_0^1 \frac{4}{1+x^2}$.
Approach: Approximate the solution by computing $\frac{1}{n}\sum_{i=0}^{n-1} \frac{4}{1+x_i^2}$ (rectangular rule), where n is the problem size parameter and $x_i = (i + \frac{1}{2})/n$ is the midpoint of the ith interval.
Note: In [12], Karp employs this problem to study parallel programming environments.
For problem sizes > machine size (2^{14}), the hand implementation of the reduction in MPL is slightly more efficient than the library function used in the Modula-2* program.

4.3.11 Paraffins Problem

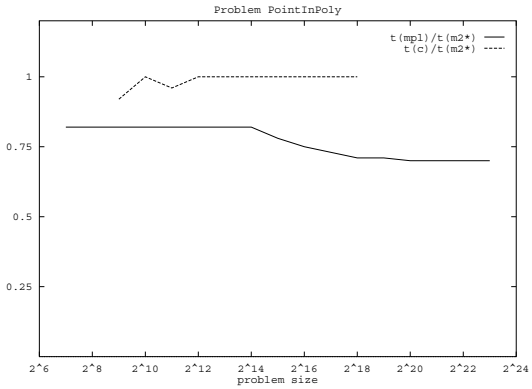
Problem: Given an integer n, output the chemical structure of all paraffin molecules for $i \leq n$, without repetition and in order of increasing size. Include all isomers, but no duplicates (from [5]).
Approach: The algorithm is partially based on [20] and has similarities to the approach used by Andrews in [5].



The scan, enumeration, and reduction library functions used in Modula-2* are more general than necessary for this problem. This generality causes performance to degrade for problem sizes > machine size.

4.3.12 Point in Polygon

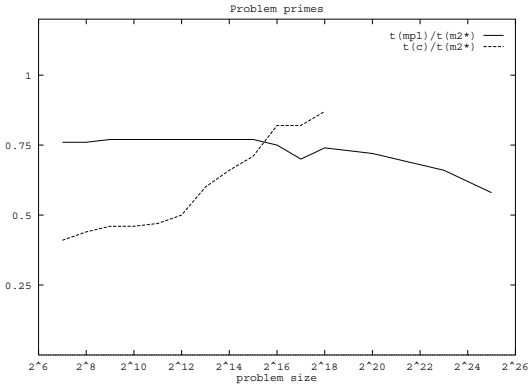
Problem: A simple polygon P with n edges and a point q are given. Determine whether the point lies inside or outside the polygon. (A polygon is simple if pairs of line segments do not intersect except at their common vertex.)
Approach: Draw a line from q that is parallel to the vertical axis. Count the number of intersections with P. The point q lies inside P if and only if this number is odd.
Note: This well-known algorithm from computational geometry appears in many text books.



See explanation for problem 4.3.10.

4.3.13 Prime Sieve

Problem: Compute all prime numbers in $[2..n]$.
Approach: We implemented the classical prime sieve. However, rather than using a virtual process per candidate, the algorithm assigns a segment of candidates to each processor. This adaptive version works much faster since division can be replaced by indexing within each segment.
Note: The problem was suggested by Hatcher [8].

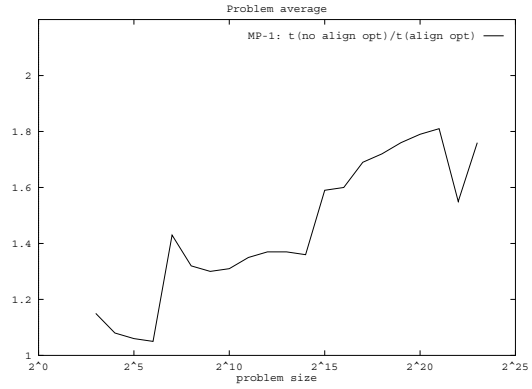


The MPL implementation of the parallel adaptive work loops can take advantage of parallel register variables. Access to them is much faster than memory access. The Modula-2* compiler does not place the same variables into registers. Hence, for growing adaptive work loops (problem size $> 2^{14}$) the performance curve degrades.

4.4 Effect of the Optimizations

Alignment and Layout

Data locality obviously pays off since data access involving communication is slower than access to local memory.



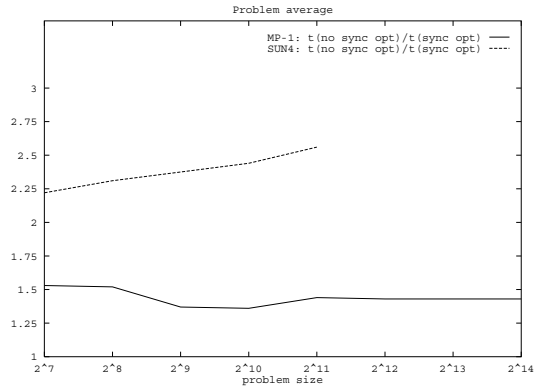
In the above diagram we compare the runtimes of two versions. The first version ($t_{no-align-opt}$) has no `ALIGNED WITH` clause in the program text. The compiler produces code that detects dynamically at runtime whether adresses are local or not. In the second version ($t_{align-opt}$), alignment optimization in the compiler has produced `ALIGNED WITH` information. The code generator thus statically knows about locality. The diagram shows the arithmetic average of the ratios over all problems. Positive errect of the alignment is indicated by the curve above unity. For example, a curve around 2 shows that the optimization halves the runtime.

On the MasPar, this optimization improves runtime performance by 40% on average. The advantage of statically determined locality grows with the amount of data accessed. No differences could be measured on a sequential workstation, since all accesses are local.

Elimination of Synchronization Barriers

The elimination obviously pays off for machines without synchronization hardware. Most MIMD machines, for example, synchronize by message passing, which can be two or three orders of magnitude slower than instruction execution. However, synchronization barrier elimination is even beneficial on SIMD machines, because it reduces virtualization overhead and the number of temporary variables needed. Furthermore, it may improve register usage.

In the following diagram, we show the performance ratio between runs without and with elimination of synchronization barriers ($t_{no-sync-opt}/t_{sync-opt}$).



Synchronization barrier elimination improves runtime by over 40% on a MasPar and by over a factor of 2 on sequential workstations. Originally, the benchmark programs had 278 synchronization barriers which were reduced to 109 by applying the optimization technique.

On SISD and MIMD machines, the performance improvement stems from the fact that fewer virtualization loops and fewer temporaries are needed. On a workstation, loop control and computation is done by the same processor. Without the elimination of synchronization barriers more than 50% of the runtime is used for loop control and memory access for additional temporaries. On the MasPar MP-1, loop control is performed by the fast frontend processor whereas the computation is done by the much slower parallel processors. Since the optimization technique only affects the frontend part the relative performance gain is smaller than that achieved on a single workstation.

5 Conclusion

We presented evidence that compilers for explicitly parallel machine-independent programs can produce competitive code. The results were obtained by comparing compiled code with hand-written and hand-optimized code. Our Modula-2* compiler presently produces code for the MasPar MP-1 series that, on average, reaches 80% of the performance of equivalent hand-coded programs. With additional optimization techniques this ratio is likely to improve even further.

High-level language compilers for parallel machines not only provide portability for parallel programs. They also simplify the task of converting sequential programs to parallel ones because the machine mapping is done by the compiler while the programmer can concentrate on finding machine-independent parallel algorithms.

A SPARC/SunOS 4.1.1 binary version of the Modula-2* compiler, the documentation, and the benchmarks are available via anonymous ftp from iraun1.ira.uka.de under pub/programming/modula2star. In order to keep track of the Modula-2* community, we ask retrievers of our Modula-2* compiler to send us their full names and addresses. Send all correspondence to msc@ira.uka.de.

References

[1] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[2] Alberto Apostoli, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. Technical Report CSD-TR-724, Purdue University, Department of Computer Sciences, May 1990.

[3] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Dordrecht, London, 1988.

[4] Ralph Butler and Ewing Lusk. *User's Guide to the p4 Parallel Programming System.* Argonne National Laboratory, 1992.

[5] John T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems.* Elsevier Science Publishers, Holland, 1992.

[6] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[7] Stefan U. Hänßgen. Msdb – a debugger, profiler and visualizer for Modula-2*. In *3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, CA, May 17–18, 1993.

[8] Philipp J. Hatcher and Michael J. Quinn. *Data-Parallel Programming on MIMD Computers.* MIT Press Cambridge, Massachusetts, London, England, 1991.

[9] Ernst A. Heinz and Michael Philippsen. Synchronization barrier elimination in synchronous FORALLs. Technical Report No. 13/93, University of Karlsruhe, Department of Informatics, April 1993.

[10] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[11] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, Mass., 1992.

[12] Alan H. Karp and R. G. Babb. A comparison of 12 parallel Fortran dialects. *IEEE Software*, 5(9):52–67, 1988.

[13] Pawel Lukowicz. Code-Erzeugung für Modula-2* für verschiedene Maschinenarchitekturen. Master's thesis, University of Karlsruhe, Department of Informatics, January 1992.

[14] MasPar Computer Corporation. *MasPar Parallel Application Language (MPL) Reference Manual*, September 1990.

[15] Michael Philippsen. Automatic data distribution for nearest neighbor networks. In *Frontiers '92: The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 178–185, Mc Lean, Virginia, October 19–21, 1992.

[16] Michael Philippsen and Markus U. Mock. Data and process alignment in Modula-2*. In Christoph W. Kessler, editor, *Automatic Parallelization – New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 171–191, AP'93 Saarbrücken, Germany, March 1–3, 1993.

[17] Michael Philippsen and Walter F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation, Salzburg, Austria, 1991*, pages 169–183. Springer Verlag, Lecture Notes in Computer Science 591, 1992.

[18] David Sankoff and Joseph B. Kruskal (eds). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Mass., 1983.

[19] Walter F. Tichy and Christian G. Herter. Modula-2*: An extension of Modula-2 for highly parallel, portable programs. Technical Report No. 4/90, University of Karlsruhe, Department of Informatics, January 1990.

[20] D. A. Turner. The semantic elegance of applicative languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 85–92, Portsmouth, NH, October 1981.

[21] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, January 1962.

[22] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.